BRIAN OVERLAND

PYTHON

WITHOUT FEAR

```
x = ['Python','is','cool']
print(' '.join(x))
```

A Beginner's Guide That Makes You Feel SMART

Python Without Fear

This page intentionally left blank

Python Without Fear

A Beginner's Guide That Makes You Feel Smart

Brian Overland

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Catalog Number: 2017946292

Copyright © 2018 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/.

ISBN-13: 978-0-13-468747-6
ISBN-10: 0-13-468747-7
1 17

For all my beloved four-legged friends:
Skyler, Orlando, Madison, Cleo, and Pogo.

This page intentionally left blank

Contents

Preface xvii

Steering Around the "Gotchas" xvii
How to Think "Pythonically" xvii
Intermediate and Advanced Features xviii
Learning in Many Different Styles xviii
What's Going on "Under the Hood" xviii
Why Python? xix

Acknowledgments xxi

Author Bio xxiii

Chapter 1 *Meet the Python* 1

A Brief History of Python 1
How Python Is Different 2
How This Book Works 3
Installing Python 4
Begin Using Python with IDLE 6
Correcting Mistakes from Within IDLE 6
Dealing with Ends of Lines 7
Additional Help: Online Sources 8

Chapter 2 *A Python Safari: Numbers* 9

Python and Numbers 9

Interlude Why Doesn't C++ Support Infinite Integers? 11

Interlude How Big Is a Google? 13

Python and Floating-Point Numbers 14

Assigning Numbers to Variables 17

Interlude What Do Python Assignments Really Do? 21

Variable-Naming Conventions in This Book 23

Some Python Shortcuts 23

Chapter 2 Summary 26

Chapter 3 *Your First Programs* 29

Temperatures Rising? 29

Interlude Python's Use of Indentation 33

Putting in a Print Message 35

Syntax Summaries 36

Example 3.1. Quadratic Equation as a Function 38

How It Works 39

Getting String Input 41

Getting Numeric Input 43

Example 3.2. Quadratic Formula with I/O 44

How It Works 45

Formatted Output String 46

Example 3.3. Distance Formula in a Script 47

How It Works 48

Chapter 3 Summary 50

Chapter 4 *Decisions and Looping* 53

Decisions Inside a Computer Program 53

Conditional and Boolean Operators 55

The if, elif, and else Keywords 56

Interlude Programs and Robots in *Westworld* 56

Example 4.1. Enter Your Age 59

How It Works 60

while: Looping the Loop 60
Example 4.2. Factorials 63
How It Works 64
Optimizing the Code 65
Example 4.3. Printing Fibonacci Numbers 67
How It Works 69
"Give Me a break" Statement 70
Example 4.4. A Number-Guessing Game 71
How It Works 72
Interlude Binary Searches and "O" Complexity 74
Chapter 4 Summary 75

Chapter 5 *Python Lists* 77

The Python Way: The World Is Made of Collections 77
Processing Lists with for 80
Modifying Elements with for (You Can't!) 82
Example 5.1. A Sorting Application 83
How It Works 84
Optimizing the Code 84
Indexing and Slicing 85
Copying Data to Slices 88
Ranges 89
Example 5.2. Revised Factorial Program 91
How It Works 91
Optimizing the Code 92
Example 5.3. Sieve of Eratosthenes 93
How It Works 94
Optimizing the Code 96
List Functions and the in Keyword 97
Interlude Who Was Eratosthenes? 98
Chapter 5 Summary 99

Chapter 6 *List Comprehension and Enumeration* 101

Indexes and the enumerate Function 101
The Format String Method Revisited 103
Example 6.1. Printing a Table 104
How It Works 105

Simple List Comprehension 106
Example 6.2. Difference Between Squares 109
How It Works 110
Interlude Proving the Equation 111
"Two-Dimensional" List Comprehension 112
List Comprehension with Conditional 114
Example 6.3. Sieve of Eratosthenes 2 115
How It Works 116
Optimizing the Code: Sets 117
Example 6.4. Pythagorean Triples 118
How It Works 119
Interlude The Importance of Pythagoras 120
Chapter 6 Summary 123

Chapter 7 *Python Strings* 125
Creating a String with Quote Marks 125
Indexing and "Slicing" 127
String/Number Conversions 130
Example 7.1. Count Trailing Zeros 131
How It Works 132
Interlude Python Characters vs. Python Strings 135
Stripping for Fun and Profit 135
Example 7.2. Count Zeros, Version 2 137
How It Works 137
Let's Split: The split Method 138
Building Strings with Concatenation (+) 139
Example 7.3. Sort Words on a Line 141
How It Works 142
The join Method 143
Chapter 7 Summary 144

Chapter 8 *Single-Character Ops* 147
Naming Conventions in This Chapter 147
Accessing Individual Characters (A Review) 148
Getting Help with String Methods 148
Testing Uppercase vs. Lowercase 149
Converting Case of Letters 150

Testing for Palindromes 151
Example 8.1. Convert Strings to All Caps 152
How It Works 153
Optimizing the Code 154
Example 8.2. Completing the Palindrome Test 154
How It Works 156
Optimizing the Code 157
Interlude Famous Palindromes 158
Converting to ASCII Code 159
Converting ASCII to Character 160
Example 8.3. Encode Strings 161
How It Works 162
Interlude The Art of Cryptography 164
Example 8.4. Decode Strings 164
How It Works 165
Chapter 8 Summary 166

Chapter 9 *Advanced Function Techniques* 167
Multiple Arguments 167
Returning More Than One Value 168
Interlude Passing and Modifying Lists 170
Example 9.1. Difference and Sum of Two Points 172
How It Works 172
Arguments by Name 173
Default Arguments 174
Example 9.2. Adding Machine 176
How It Works 176
Optimizing the Code 177
Importing Functions from Modules 178
Example 9.3. Dice Game (Craps) 179
How It Works 180
Interlude Casino Odds Making 182
Chapter 9 Summary 185

Chapter 10 *Local and Global Variables* 187
Local Variables, What Are They Good For? 187
Locals vs. Globals 188
Introducing the global Keyword 190

The Python "Local Variable Trap" 190
Interlude Does C++ Have Easier Scope Rules? 191
Example 10.1. Beatles Personality Profile (BPP) 192
How It Works 195
Example 10.2. Roman Numerals 196
How It Works 197
Optimizing the Code 198
Interlude What's Up with Roman Numerals? 200
Example 10.3. Decode Roman Numerals 201
How It Works 202
Optimizing the Code 203
Chapter 10 Summary 204

Chapter 11 *File Ops* 207
Text Files vs. Binary Files 207
The Op System (os) Module 208
Interlude Running on Other Systems 211
Open a File 211
Let's Write a Text File 213
Example 11.1. Write File with Prompt 214
How It Works 214
Read a Text File 216
Files and Exception Handling 217
Interlude Advantages of try/except 219
Example 11.2. Text Read with Line Numbers 220
How It Works 221
Other File Modes 223
Chapter 11 Summary 224

Chapter 12 *Dictionaries and Sets* 227
Why Do We Need Dictionaries, Ms. Librarian? 227
Adding and Changing Key-Value Pairs 229
Accessing Values 230
Searching for Keys 231
Interlude What Explains Dictionary "Magic"? 232
Example 12.1. Personal Phone Book 232
How It Works 234

Converting Dictionaries to Lists 235
Example 12.2. Reading Items by Prefix 236
How It Works 238
Example 12.3. Loading and Saving to a File 238
How It Works 240
All About Sets 241
Operations on Sets 242
Interlude What's So Important About Sets? 244
Example 12.4. Revised Sieve of Eratosthenes 244
How It Works 245
Chapter 12 Summary 246

Chapter 13 *Matrixes: 2-D Lists* 249
Simple Matrixes 249
Accessing Elements 250
Irregular Matrixes and Length of a Row 251
Multiplication (*) and Lists 252
The Python Matrix Problem 253
How to Create N*M Matrixes: The Solution 254
Interlude Why Isn't It Easier? 255
Example 13.1. Multiplication Table 256
How It Works 257
Example 13.2. User-Initialized Matrix 258
How It Works 259
Optimizing the Code 260
How to Rotate a Matrix 261
Interlude Pros and Cons of Garbage Collection 263
Example 13.3. Complete Rotation Example 264
How It Works 266
Optimizing the Code 267
Chapter 13 Summary 268

Chapter 14 *Winning at Tic-Tac-Toe* 271
Design of a Tic-Tac-Toe Board 271
Plan of This Chapter 273
Phase 1 273
Phase 2 273
Phase 3 273

Python One-Line if/else 274
Example 14.1. Simple Two-Player Game 274
How It Works 276
Interlude Variations on Tic-Tac-Toe 279
The count Method for Lists 279
Example 14.2. Two-Player Game with Win Detection 279
How It Works 282
Introducing the Computer Player 285
Example 14.3. Computer Play: The Computer Goes First 287
How It Works 290
Playing Second 291
Interlude The Art of Heuristics 292
Chapter 14 Summary 294

Chapter 15 *Classes and Objects I* 295
What's an Object? 295
Classes in Python 296
How Do I Define a Simple Class? 297
How Do I Use a Class to Create Objects? 297
How Do I Attach Data to Objects? 298
How Do I Write Methods? 300
The All-Important __init__ Method 301
Interlude Why This self Obsession? 302
Design for a Database Class 303
Interlude C++ Classes Compared to Python 304
Example 15.1. Tracking Employees 305
How It Works 307
Defining Other Methods 309
Design for a Point3D Class 310
Point3D Class and Default Arguments 312
Three-Dimensional Tic-Tac-Toe 312
Example 15.2. Looking for a 3-D Win 313
How It Works 314
Example 15.3. Calculating Ways of Winning 315
How It Works 317
Optimizing the Code 317
Chapter 15 Summary 318

Chapter 16 *Classes and Objects II* 321

Getting Help from Doc Strings 321
Function Typing and "Overloading" 323
Interlude What Is Duck Typing? 325
Variable-Length Argument Lists 326
Example 16.1. PointN Class 327
How It Works 329
Optimizing the Code 330
Inheritance 331
The Fraction Class 333
Example 16.2. Extending the Fraction Class 334
How It Works 335
Class Variables and Methods 337
Instance Variables as "Default" Values 339
Example 16.3. Polygon "Automated" Class 340
How It Works 342
Interlude OOPS, What Is It Good For? 343
Chapter 16 Summary 344

Chapter 17 *Conway's Game of Life* 347

Interlude The Impact of "Life" 347
Game of Life: The Rules of the Game 348
Generating the Neighbor Count 350
Design of the Program 352
Example 17.1. The Customized Matrix Class 352
How It Works 353
Moving the Matrix Class to a Module 354
Example 17.2. Printing a Life Matrix 355
How It Works 355
The Famous Slider Pattern 358
Example 17.3. The Whole Game of Life Program 358
How It Works 360
Interlude Does "Life" Create Life? 363
Chapter 17 Summary 364

Chapter 18 *Advanced Pythonic Techniques* 367
Generators 367
Exploiting the Power of Generators 369
Example 18.1. A Custom Random-Number Generator 370
How It Works 372
Interlude How Random Is "Random"? 373
Properties 375
Getter Methods 376
Setter Methods 377
Putting Getters and Setters Together 378
Example 18.2. Multilevel Temperature Object 379
How It Works 380
Decorators: Functions Enclosing Other Functions 382
Python Decoration 385
Example 18.3. Decorators as Debugging Tools 387
How It Works 388
Chapter 18 Summary 389

Appendix A *Python Operator Precedence Table* 391

Appendix B *Summary of Most Important Formatting Rules for Python 3.0* 393
1. Formatting Ordinary Text 393
2. Formatting Arguments 393
3. Specifying Order of Arguments 393
4. Right Justification Within Field of Size N 394
5. Left Justification Within Field of Size N 394
6. Truncation: Limit Size of Print Field 394
7. Combined Truncation and Justification 395
8. Length and Precision of Floating-Point Numbers 395
9. The Padding Character 395

Appendix C *Glossary* 397

Index 407

Preface

There's a lot of free programming instruction out there, and much of it's about Python. So for a book to be worth your while, it's got to be good…it's got to be really, really, *really* good.

I wrote this book because it's the book I wish was around when I was first learning Python a few years back. Like everybody else, I conquered one concept at a time by looking at almost a dozen different books and consulting dozens of web sites.

But this is Python, and *it's not supposed to be difficult!*

The problem is that not all learning is as easy or fast as it should be. And not all books or learning sites are *fun*. You can, for example, go from site to site just trying to find the explanation that really works.

Here's what this book does that I wish I'd had when I started learning.

Steering Around the "Gotchas"

Many things are relatively easy to do in Python, but a few things that ought to be easy are harder than they'd be in other languages. This is especially true if you have any prior background in programming. The "Python way" of doing things is often so different from the approach you'd use in any other language, you can stare at the screen for hours until someone points out the easy solution.

Or you can buy this book.

How to Think "Pythonically"

Closely related to the issue of "gotchas" is the understanding of how to *think* in Python. Until you understand Python's unique way of modeling the world,

you might end up writing a program the way a C programmer would. It runs, but it doesn't use any of the features that make Python such a fast development tool.

```
a_list = ['Don\'t', 'do', 'this', 'the' ,'C', 'way']
for x in a_list:
    print(x, end=' ')
```

This little snippet prints

```
Don't do this the C way
```

Intermediate and Advanced Features

Again, although Python is generally easier than other languages, that's not universally true. Some of the important intermediate features of Python are difficult to understand unless well explained. This book pays a lot of attention to intermediate and even advanced features, including list comprehension, generators, multidimensional lists (matrixes), and decorators.

Learning in Many Different Styles

In this book, I present a more varied teaching style than you'll likely find elsewhere. I make heavy use of examples, of course, but sometimes it's the right conceptual figure or analogy that makes all the difference. Or sometimes it's working on exercises that challenge you to do variations on what's just been taught. But all of the book's teaching styles reinforce the same ideas.

What's Going on "Under the Hood"

Although this book is for people who may be new to programming altogether, it also caters to people who want to know how Python works and how it's fundamentally different "under the hood." That is, how does Python carry out the operations internally? If you want more than just a simplistic introduction, this book is for you.

Why Python?

Of course, if you're trying to decide between programming languages, you'll want to know why you should be using Python in the first place.

Python is quickly taking over much of the programming world. There are some things that still require the low-level capabilities of C or C++, but you'll find that Python is a *rapid application development tool*; it multiplies the effort of the programmer. Often, in a few lines of code, you'll be able to do amazing things.

More specifically, a program that might take 100 lines in Python could potentially take 1,000 or 2,000 lines to write in C. You can use Python as "proof of concept": write a Python program in an afternoon to see whether it fulfills the needs of your project; then after you're convinced the program is useful, you can rewrite it in C or C++, if desired, to make more efficient use of computer resources.

With that in mind, I'll hope you'll join me on this fun, exciting, entertaining journey. And remember this:

```
x = ['Python', 'is', 'cool']
print(' '.join(x))
```

Register your copy of *Python Without Fear* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780134687476) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

Acknowledgments

It's customary for authors to write an acknowledgments page, but in this case, there's a particularly good reason for one. There is no chapter in this book that wasn't strongly influenced by one of the collaborators: retired Microsoft programmer (and software development engineer) John Bennett.

John, who has used Python for a number of years—frequently to help implement his own high-level script languages—was particularly helpful in pointing out that this book should showcase "the Python way of doing things." So the book covers not just how to transcribe a Python version of a C++ solution but rather how to take full advantage of Python concepts—that is, how to "think in Python."

I should also note that this book exists largely because of the moral support of two fine acquisition editors: Kim Boedigheimer, who championed the project early on, and Greg Doench, whom she handed the project off to.

Developmental and technical editors Michael Thurston and John Wargo made important suggestions that improved the product. My thanks go to them, as well as the editorial team that so smoothly and cheerfully saw the manuscript through its final phases: Julie Nahil, Kim Wimpsett, Angela Urquhart, and Andrea Archer.

This page intentionally left blank

Author Bio

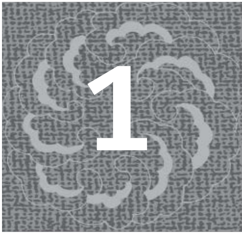
At one time or another, Brian Overland was in charge of, or at least influential in, documenting all the languages that Microsoft Corporation ever sold: Macro Assembler, FORTRAN, COBOL, Pascal, Visual Basic, C, and C++. Unlike some people, he wrote a lot of code in all these languages. He'd never document a language he couldn't write decent programs in.

For years, he was Microsoft's "go to" man for writing up the use of utilities needed to support new technologies, such as RISC processing, linker extensions, and exception handling.

The Python language first grabbed his attention a few years ago, when he realized that he could write many of his favorite applications—the Game of Life, for example, or a Reverse Polish Notation interpreter—in a smaller space than any computer language he'd ever seen.

When he's not exploring new computer languages, he does a lot of other things, many of them involving writing. He's an enthusiastic reviewer of films and writer of fiction. He's twice been a finalist in the Pacific Northwest Literary Contest.

This page intentionally left blank



Meet the Python

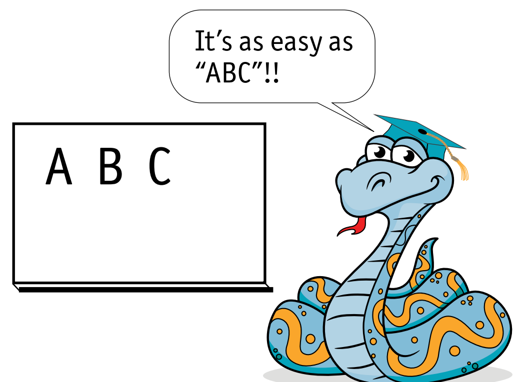
What if I told you there's a computer language that's easier to learn, easier to get started with, and easier to accomplish a great deal with, using only a few lines of code, than other computer languages?

In the opinion of millions, Python is that language. Derived from a language called ABC (as in "simple as ABC"), it's gained a massive worldwide following over the last two decades. So many programmers have joined the Python community that there are more than 100,000 free packages that work with the basic Python setup.

Come join the Python stampede. In this book I show you how to get started even if you have limited programming experience. I also steer you around the "gotchas"—the things Python does so differently that they trip up experienced programmers. This book is for new programmers as well as experienced programmers alike, and it discusses what goes on under the covers.

A Brief History of Python

Python was invented in 1991 by Dutch programmer Guido van Rossum, who derived much of it from the ABC language (not to be confused with C).



ABC had many features that exist today in Python. Van Rossum, whose title in the Python world is Benevolent Dictator for Life (BDFL), also incorporated elements of the Modula-3 language.

Van Rossum named the language after the BBC comedy series *Monty Python's Flying Circus*, so the connection to pythons is indirect, although troupe member John Cleese originally came up with "Python" as suggesting something "slithering and slimy" (source: Wikipedia.org). So there you have it—there is a connection to reptiles after all.

Since then, several versions of Python have been developed, adding important capabilities, the latest of which is Python 3.0. This book uses Python 3.0, although it includes notes about adapting examples to Python 2.0.

How Python Is Different

The first thing to know about Python is that Python is free.

Many Python extensions are free and come with the basic download. These modules offer features such as math, date/time, fractions, randomization, and tkinter, which supports a graphical user interface that runs across multiple platforms. Again, all are free.

Python's built-in numeric support is impressive, as it includes complex numbers, floating-point, fractions (from the Fractions module), and "infinite integers."

Python has attracted an extraordinary following. Many developers provide libraries—called *packages*—to their fellow Python programmers, mostly free of charge. You can get gain access by searching for *Python Package Index* in your Internet browser and then going to the site. As of this writing, the site offers access to more than 107,000 packages.

At first glance, a Python program may look something like code in other languages, but a close look reveals major differences.

- Unlike most languages, Python has no "begin block" or "end block" syntax—all relationships are based on indentation! Although this might seem risky to a C programmer, it enforces a consistent look that's more comprehensible to beginners.
- Python has no variable declarations. You create variables by assigning values to them. This goes a long way toward simplifying the language syntactically, but it also creates hidden "gotchas" at a deep level. This book will steer you around them.
- Python is built heavily on the idea of *iteration*, which means looping through sequences. This concept is built deeply into high-level structures (lists, dictionaries, and sets). Use them well, and you'll be able to get a great deal done in a small space.

Python is often considered a "prototyping" or "rapid application development" language because of these abilities. You can write a program quickly in Python. If you later want to improve machine-level efficiency, you can later rewrite the program in C or C++.

How This Book Works

I believe strongly in learning by example as well as by theory. The plan of this book is to teach the basics of Python (as well as some intermediate and advanced features) by doing the following:

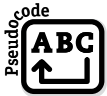
- Introducing a Python feature, using syntax diagrams and short examples
- Showing a major example that demonstrates the practical application of the feature
- Including a "How It Works" section that deconstructs the example code line by line
- Listing a set of exercises that challenge you to do variations on the example

Because Python has an interactive development environment, IDLE, I often invite you to follow along with the shorter examples, as well.

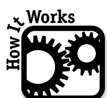
This book uses a number of icons in the margin to help give you additional visual cues.

These sections describe some basic rule of Python syntax. Anything meant to be entered at the keyboard precisely as shown (such as a keyword or punctuation) is in bold. Meanwhile, placeholders, which contain text you supply yourself, are in italics. For example, in the syntax for the **global** statement, the keyword itself is in bold, while the name of the variable—which you supply—is in italics.

global *variable_name*



This icon indicates a block of pseudocode, which systematically describes each step of a program purely in English, not Python-ese. However, because Python statements are often not so far from English, I don't always need to use pseudocode. It can still be helpful, on occasion, for summarizing program design.



This icon indicates a section that deconstructs every line of a major example, or at least every line that isn't already trivial.

This icon indicates a section that provides exercises based on the preceding example. You'll learn Python much faster if you try at least some of the exercises.

This icon precedes a section that shows how to revise or greatly improve an example. These are not included for every example. Where this book does use them, it's because the example used the more obvious way to do something; the "optimized" approach will then show how the more experienced, sophisticated Python programmer would handle the job.

Installing Python

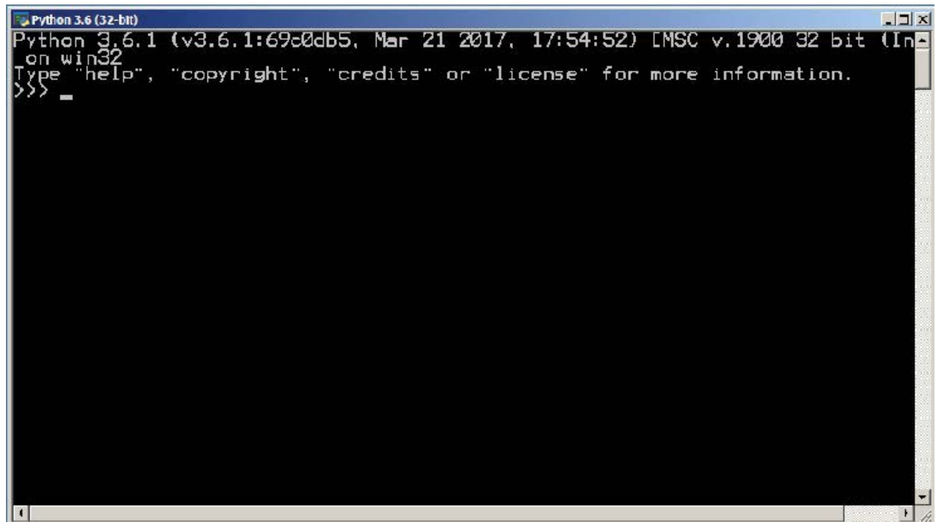
The steps for installing Python are essentially the same regardless of whether you have a Windows-based system, Macintosh, or any other system that Python supports. Here are the basic steps:

1. Go to the Python home page: **python.org**.
2. Open the Downloads menu.
3. If a Downloads for Windows screen appears, click the Python 3.6.1 button. If your system does not run Windows, you'll need to select another operating system by examining all the choices in the Downloads menu.
4. Click the Save File button.
5. Find the file you just saved; any system will generally have a place that it puts downloads. This saved file contains the Python installer. Double-click the name of this saved file and follow the instructions.

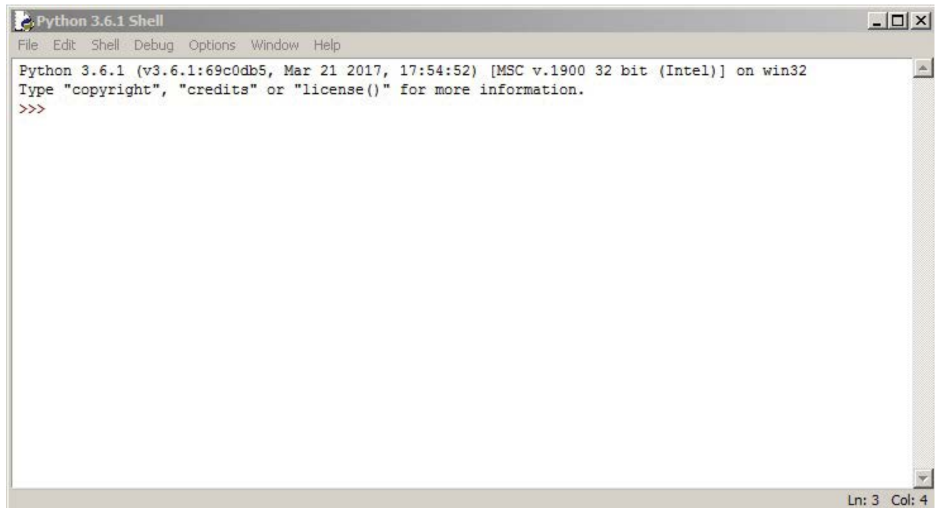
If all goes well, Python is installed on your computer with all the basic modules, tkinter (GUI development) included. Now you have a choice to make. To start using Python, you can use "basic interactive mode"—which is functional but nothing special—or you can use IDLE, the interactive development environment.

I strongly recommend the latter. IDLE does everything the basic interactive mode does, and a great deal more. In the next section, I describe some ways of using IDLE that can save you a lot of time later.

Here's what basic interactive mode looks like. It offers only rudimentary editing and no support for loading programs from text files.



Here's what IDLE looks like. Notice all the menus it provides. You can do a great deal more—including loading programs from text files and debugging them—than you can with the basic interactive mode.



From within Windows, you should find the basic interactive-mode application right on the Start menu. But this is not the Python you want. It's well worth your while to select Programs, select Python, and then finally select IDLE.

With Mac systems (assuming you have downloaded Python, including IDLE), you may need to get to IDLE by opening Finder and selecting Applications; then select Python and finally select IDLE. Your download may or may not even have basic mode.

Begin Using Python with IDLE

Start IDLE, the interactive development environment. I advise you to make this your home base, the place you'll want to spend most of your time while learning Python. You should use your system to put the icon on your desktop so that you can easily start IDLE any time you want.

As soon as you start IDLE, you'll see a prompt, like this:

```
>>>
```

In response to this prompt, you can enter a Python statement or expression. You can also get help by typing the **help** command followed by a type name, like this:

```
>>>help(str)
```

Here I've shown the user input—the characters you would enter at the keyboard—in bold; output from Python is in normal font. I follow this convention throughout the book.

Correcting Mistakes from Within IDLE

One of the best features of IDLE is that it makes error correction easy. Let's say you sit down and enter the following:

```
>>>x = z
```

As you'll learn in upcoming chapters, this assignment statement produces an error if the variable z has not already been assigned a value. The environment responds by printing a message like this:

```
Traceback (most recent call last):
  File "<pyshell#205>", line 1, in <module>
    x = z
NameError: name 'z' is not defined
```

In this case, it's easy to reenter the offending statement. But suppose you have a much longer block of code that is erroneous and you don't want to retype the whole thing. Here's an example:

```
def print_nums(n):
    i = 1
    while i <= n:
        print(i, end='\t')
        i +++= 1
```

The problem with this block of code is that it ends with the line `i +++= i` instead of `i += i`. There was supposed to be only one plus sign (+).

You'd like to fix this error but don't want to retype all those statements. Fortunately, Python makes error correction easy. Just do the following:

1. Position the cursor on any line in the block of code. (If the block of code is only one line, make sure the cursor is at the end of the line.)
2. Press the Enter key.

Voilà! The entire block of code magically reappears, with the cursor positioned at the end; you can then fix whatever you need to fix. Use the arrow keys to go back to any statement and then make your corrections. Finally, to resubmit a block of code, place the cursor at the end of the last line again and press Enter twice.

Remember this technique. It will save you many hours of work.

Dealing with Ends of Lines

Because of the way in which Python interprets lines, you cannot freely cross physical-line borders as you can in C. But what if you need to enter an exceptionally long line?

The end of a physical line usually terminates a Python statement, because there is no statement-terminator syntax as in C. However, an open parenthesis, curly brace, or bracket automatically continues the virtual line to the next physical line. Here's an example:

```
total_amount = (this_amount + that_amount
   + a_big_number + count + even_more amounts )
```

The open parenthesis, (, on the top line creates a situation in which you can freely continue the statement onto other lines, until this parenthesis is matched. This is one case in which indentation doesn't matter but is only for readability. (Usually, Python forces indentation to be consistent.)

Occasionally, you may not be able to rely on this technique. If you really need to continue a physical line and have no alternative, you can use a backslash.

```
>>>my_str = 'I am typing a very long \
line of code.'
```

This example raises a question: How would you type a literal backslash in quoted string? The answer is that you'd use a double backslash, \\, to represent a literal backslash.

```
>>>my_str = 'I am typing a backslash: \\ \
in a long line of code.'
```

Chapter 7, "Python Strings," will get into the details of creating quoted strings in much greater detail.

In the last few pages, I've given you some Python survival skills. Now, if you're ready, it's time to go on a Python safari.

Additional Help: Online Sources

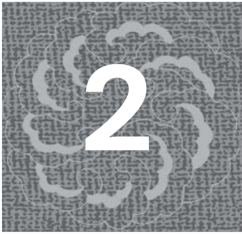
In this chapter, I've strongly suggested you download Python 3.0 or newer. If you are using an older version of Python 2.0, most of the code in this book will work, but you may need to make some adjustments. Although I've included some version notes, you can find additional help on the following websites:

wiki.python.org/moin/Python2orPython3

wiki.python.org/moin/PortingToPy3k/BilingualQuickRef

Although many chapters in this book feature examples that are relatively short and therefore easy to type in yourself, some of the later chapters feature longer program listings. You may find it helpful to download the code. You can find the code listings at this site:

brianoverland.com/books



2 A Python Safari: Numbers

Now that you've installed Python (you did install it, didn't you?), you're ready to go. The IDLE interactive environment is your starting point for Python safaris. But it's not just for beginners; for a long time, you should find it useful as a learning device as you advance.

But now—let's start! This chapter covers the following:

- Python "infinite" integers
- Integer vs. floating-point operations
- How variables are used in Python

Python and Numbers

Start IDLE. Up should come a prompt, although you may have to hit Enter one time to get it to appear.

```
>>>
```

Type in your favorite number and press Enter again. Let's say it's five.

```
>>>5
5
```

Here I use bold—as I will throughout most of this book—to show user input. Entering **5** and getting **5** in response is not that exciting. But let's do a calculation.

```
>>>10 + 15
25
```

So…interactive Python is a handy calculator! It's obvious what + does. But as with other programming languages, we can throw in subtraction (–) and multiplication (*).

```
>>>25 - 5 * 2
15
```

There's an issue of *precedence* here. If you subtract first, you get a final result of 40. But standard conventions of math say you multiply first, so 15 is correct. We could've used parentheses to get a different result.

```
>>>(25 - 5) * 5
40
```

So far, Python just appears to be a convenient one-line calculator. But wait. The next thing I want to show you is that Python—almost (but not quite) uniquely among programming languages—can handle extraordinarily large numbers.

The easiest way to generate super-large numbers is to use the exponent operator. This operator consists of two asterisks (**) in a row. For example, taking 3 to the 4th power gives us 3 times 3 times 3 times 3, or 81.

```
>>>3 ** 4
81
```

Eighty-one isn't that big, but 9 to the 30th power is 30 factors of 9 multiplied together…and that's a pretty big number.

```
>>>9 ** 30
42391158275216203514294433201
```

You might be tempted to say, "I've used floating-point numbers, and they can get bigger than this, so what's so impressive?"

In Python, as in most other programming languages, integers are absolutely precise, which means that no matter how high they get, adding 1 always creates a new value. That is not true of floating-point numbers, as you'll see.

For example, enter **10**40** and **10**40+1** and see what happens.

```
>>>10 ** 40
10000000000000000000000000000000000000000
>>>10 ** 40 + 1
10000000000000000000000000000000000000001
```

You should be able to see that these two numbers are distinct; they are not equal. Just look at the last digit in each number and compare. If you apply the test-for-equality operator, you can verify this directly in Python.

Test-for-equality (==) is an operator you'll probably use a great deal in Python programming. This operator produces the value **True** or **False**, which are special reserved words, called *keywords*. That means they have a special predefined meaning to the Python language.

If the two expressions produce indistinguishable results, then this operator will produce **True**.

```
>>>10 ** 40 == 10 ** 40 + 1
False
```

The result, **False**, means that Python recognizes that the results on either side of the equal signs (==) are not equal. So, adding 1 does produce a truly distinct number.

That, in turn, is amazing, because it shows the usefulness of super-large integers; they never lose their precision. Even if you're counting a very large population, one item at a time, you can rest assured that each time you add 1, you get a distinct value.

This Python "infinite integer" feature is impressive. You even can handle—with absolute precision—some legendary large numbers. For example, you may have heard of the term *google* (also spelled *googol*). In math, a google (or googol) is 10 to the 100th power, an almost unimaginably large number.

But it's a piece of cake to handle this number in Python.

```
>>>10 ** 100
10000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
```

You can start with this number and then count forward one at a time. As before, adding 1 produces a distinct number, because there's no loss of precision.

```
>>>10 ** 100 + 1
10000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000001
```

By the way, there is a subtlety here. In the absence of parentheses, the exponent operator is applied before addition. There is a definite precedence to Python operators, as listed in Appendix A.

Interlude

Why Doesn't C++ Support Infinite Integers?

Among programming languages in wide use today, Python is the only language that supports "infinite integers" in its standard form. Why doesn't C++ support this feature?

▼ *continued on next page*

Interlude

▼ *continued*

Using the class feature of C++, which is very powerful, you could in fact create an infinite-integer class for yourself. My book *C++ Without Fear* doesn't discuss how to do that exactly, but by using all the tools in that book, you could figure it out.

But it would not be easy. Creating a super-integer class that supports addition and subtraction would not be so difficult, but multiplication and division are tough. The Python integer type was created by math specialists who understand optimal ways to multiply and divide exceptionally large numbers. The good news is that when you use Python, all these problems have been solved for you.

Although Python's integer capacity is impressive, there is ultimately a limitation—imposed not by Python but by the computer's hardware capabilities. This is a fuzzy limitation, admittedly. You can go much bigger than a google, specifying (for example) 10 to the 200th power.

```
>>>10 ** 200
1000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000
```

This is a number so large it is beyond the ability of the human mind to grasp. Yet Python can handle this number.

Mathematicians can think of far higher numbers still. A *google-plex* is 10 raised to the power of a google. This is easy enough to specify.

```
10 ** (10 ** 100)
```

But as they say on television, don't try this at home! Printing out such a number would be *1 followed by a google zeros!* Just the number of zeroes the computer would have to print would be larger than the number of atoms in the universe, and that far exceeds the ability of your computer to print zeroes.

Before we leave "the google" altogether, consider that we can use the abilities in this section to solve problems *not even solvable in most programming languages*—that is, not without great difficulty. Consider the problem of taking the number google plus 1 and then determining whether it is divisible by 7.

The remainder operator (%), which has the same precedence as multiplication and division, comes to our aid here. This operator produces the remainder after division; if a number is evenly divisible by 7, then—after division by 7—it will produce a remainder of 0. Let's try this.

```
>>>(10 ** 100 + 1) % 7
5
```

What did we just learn? The first number bigger than a google (which is a google plus 1) yields a remainder of 5 if you divide by 7. It is therefore not a multiple of 7. With a little mathematical reasoning, you can quickly infer that the smallest integer bigger than a google, which is a multiple of 7, is a google plus 3.

That is a fact that would be difficult or impossible to determine with other programming languages.

EXERCISES

Exercise 2.1.1. Would you expect the power operator (**) to take precedence over multiplication? Try a calculation to test your guess.

Exercise 2.1.2. Use Python to generate the result of 7 to the 40th power.

Exercise 2.1.3. How big, precisely, is the address space of a 64-bit architecture computer? Bear in mind that for each additional bit, the address space doubles. Use Python to generate this number.

Exercise 2.1.4. Use Python to help determine the first number bigger than a google that is divisible by 13. You may need to use a little trial and error, but you shouldn't need too much.

How Big Is a Google?

The best estimates by scientists now say the number of elementary particles in our physical universe—counting all electrons, protons, neutrons, and so on—is around 10 to the 80th power.

The number of grains of sand on our planet has been estimated at a "mere" 7.5 times 10 to the 18th power. Therefore, the number of particles in the universe is (no surprise!) incomprehensibly larger.

Although 10 to the 80th is pretty big, it still falls short of a google by 10 to the 20th, and 10 to the 20th is 1 followed by 20 zeroes. That number itself is not so small.

```
10,000,000,000,000,000,000,000
```

That is to say, 10,000 times a billion times another billion. What Carl Sagan called "billions and billions." So, if every universe was like our own, it would take all the particles *in that many universes* to equal a google!

▼ continued on next page

▼ *continued*

The base number here—10 to the 80th power—is just a few powers of 10 short of the estimated size of the physical universe in cubic centimeters. This follows from the estimate that the diameter of the physical universe is 10 to the 26th meters across.

But as far as mathematicians are concerned, we've hardly gotten started. The number called a *google-plex* can be expressed as 1 *followed* by a google's worth of zeroes. Therefore, just to write out the number using standard notation would be 1 followed by so many zeroes they could not fit into the universe, if each 0 was written on its own little block a 10th of a meter (roughly 3 inches) in diameter.

It would take "billions and billions" of physical universes like our own just to find space to write down all the zeroes in a google-plex!

Fortunately, scientific notation and substitution makes it possible to write down these figures, even though they are so large as to be far beyond corresponding to anything in the universe.

```
google = 10 ** 100
google-plex = 10 ** google
```

As I've shown in this chapter, Python is good at handling numbers in the range of a google or even a google squared—which would be 1 followed by 200 zeroes. Even division between two such quantities is fast and efficient. But don't ask Python to try to deal with a google-plex, which is far beyond the ability of Python to handle. It's really only comprehensible as a theoretical notion.

Oh, and the physical constant that comes closest to a google? That would be the density of the universe (in kilograms per cubic meter) at the time of the Big Bang—or rather one unit of Planck time immediately after the Big Bang. That number is 10 to the 96th power, and it still comes up short.

Python and Floating-Point Numbers

Another operation, of course, is division. Division is special, because even though you use two integers (an integer being a number with no fractional portion), division has the possibility of producing a fractional result; if it is fractional, it will be stored in floating-point format.

```
>>>15 / 2
7.5
```

Unfortunately, here is where version differences raise their ugly head. This is the result you can expect to see with Python version 3.*x*. With version 2.*x*, if two

integers are involved in division, the result is automatically rounded down to the nearest integer. To get the same effect with version 3.*x*, use integer division (//).

```
>>>15 // 2
7
```

This looks like the remainder is being thrown away. Indeed it is. But you can always use the remainder-division operator (%) to get that quantity.

```
>>>15 % 2
1
```

Version ▶ In version 2.*x*, all division between two integer operands is interpreted as integer, or rather "ground" division. That is to say, if both operands are integers, division will throw fractional portions away. Sadly, Python 3.*x* is not 100 percent backward compatible, and integer division is one of those areas in which there is a significant difference.

With version 2.*x*, to force division to be precise, you'd need to promote one of the operands to floating point, either by specifying it in floating-point format (such as "2.0") or by using a **`float`** conversion.

```
>>>15 / float(2)
7.5
```

◀ Version

For the most part, you don't need to worry about the details of how the computer carries out floating-point math. However, there are a few things you do need to know.

First, floating-point numbers have the capacity to represent fractions. For that reason, there are many situations in which you'll want to use floating point.

Second, to specify floating-point format, just use a decimal point. The following numbers are both considered floating point by Python, even though the second case contains a zero fractional portion:

```
>>>1.75
1.75
>>>9.0
9.0
```

The third thing to understand about floating-point format is that you can freely combine integer and floating-point expressions. Python will happily promote an integer expression to floating point so that the numbers can be freely combined.

```
>>>1 + 2.5
3.5
```

Floating-point numbers, unlike integers, have limited precision. This means that very large floating-point numbers lose the ability to distinguish between one number and the next. Consider the following number, formed by taking 9 to the 30th power—but doing so with floating-point math, not integer:

```
>>>9.0 ** 30
4.23911582752162e+28
```

We used a decimal point in 9.0, so the expression is treated as floating point, not integer. With large floating-point numbers (or tiny amounts extremely close to 0.0), Python switches to scientific notation. The number shown here is approximately 4.239 times 10 to the 28th power.

If you now add 1 (either floating-point or integer) to this result, you'll see that there's a limited precision. The following produces a result that isn't distinguishable from the previous result:

```
>>>9.0 ** 30 + 1
4.23911582752162e+28
```

There are cases where you might want to use this number in counting, and in such cases, it's critical that adding 1 produces a new number. So if we test the two quantities for equality (==), we should get **False**. But look what happens:

```
>>>9.0 ** 30 == 9.0 ** 30 + 1.0
True
```

In other words, we added 1 to the quantity 9.0 ** 30 and failed to get a new number.

Think about what this means. We added 1 to a number, which should produce a different number not equal to the first! Yet Python says they are equal. This means either that Python doesn't understand math or that there was a rounding error because of loss of precision.

As we saw in the previous section, adding 1 to the quantity 9 ** 30 (which is an integer expression, not floating point) *does* produce a new number. That's because integers, unlike floating-point numbers, are always precise.

The moral of the story is, if you have a number that's used for counting or indexing purposes, it should be an integer.

EXERCISES

Exercise 2.2.1. Describe in English the meaning of the expression `5.23911582752162e+22`.

Exercise 2.2.2. Based on how the expressions were evaluated in the previous examples, what would you say is the precedence of test-for-equality (==) relative to arithmetic operators (+, –, /, *, **)?

Assigning Numbers to Variables

So far, we've been using Python as a super-powered calculator, able to handle numbers such as a google.

But programming requires variables. A variable is simply a name to which we assign a data value. In Python, any variable can refer to any type of data at any time. That's because Python variables have no type; only data objects do. I'll get into the consequences of that fact later.

It's easy to start using variables in Python. For example, you can enter the following:

```
>>>a = 1
>>>b = 2
>>>a + b
3
```

If you've used any other programming language before—or even if you haven't—what happened here should be clear. Even if this is your first attempt at programming, this should still be easy to understand. Here's what we did:

1. Associate the variable name `a` with the value 1.
2. Associate the variable name `b` with the value 2.
3. Add `a` and `b` together, which represent 1 and 2, respectively. Python responds as if **`1 + 2`** were entered.

Although Python may seem lax, there are restrictions. The third statement in this example used `a + b` on the right side of the assignment; but this was valid only because `a` and `b` had already been created. Here is the general rule, and it's the most fundamental rule in Python:

A variable must be created before being used, but an assignment (=) creates a variable if it does not already exist.

One upshot of this rule is that—with few exceptions—a variable must appear on the *left* side of an assignment before it appears on the right.

For example, you can create a variable named `my_amount`, but if a new variable appears on the *right* of the assignment, that's an error. Here, the use of `x` on the right side causes an error:

```
my_amount = x          # Error! x not yet created.
```

The problem was that this statement used `x`, even though `x` didn't represent anything. The solution is to create x first, by assigning it a value. Only then can `x` be used in any other context.

```
>>>x = 10
>>>my_amount = x
```

Python has no trouble with these statements now. The effect in this case is to associate both of the names, `x` and `my_amount`, with the data value, 10.

What happens if we assign a value to a variable a second time? The answer is

- First, the value on the right is fully calculated.
- Second, any previous association the variable had is now canceled.
- The variable is now associated with the value on the right side of the equal sign (=).

Once again, it doesn't matter whether the variable previously referred to integer or floating-point data; the variable becomes associated with the new value. Here's an example:

```
>>>x = 7.59
>>>x
7.59
>>>x = 2
>>>x
2
```

A variable may appear on *both* sides of an assignment—but only if it was previously created by another assignment. The old value is used in the calculation of the new. Here's an example:

```
>>>n = 5
>>>n = n + 1
>>>n
6
```

Another rule is that every name in the Python language is case-sensitive. Consequently, the following produces 101, not 200!

```
>>>a = 1
>>>A = 100
>>>a + A
101
```

Assignment, in Python, is a statement. That means there's a strict syntax that determines how you can use it. With some exceptions we'll cover later, this is how you use assignment:

```
variable_name = expression
```

Remember that a single equal sign (=) is used here, not double (==), which differentiates assignment from test-for-equality.

As for variable names, rules are as follows:

- The first letter in a variable (or other symbolic name) must be an underscore (_) or a letter.
- The other characters may be any combination of underscores, letters, and numerals.

The expression on the right can be a single value, or it can be more complex. Here's an example:

```
>>>my_num1 = 7
>>>my_num2 = my_num1 + (3.0 / 2)
>>>my_num2
8.5
```

The first line here creates the variable `my_num1`, through assignment. The second statement creates the variable `my_num2` while using `my_num1` on the right; this usage of `my_num1` is valid because `my_num1` was already created. The next line is just the name `my_num2`. When you are in the interactive environment (IDLE) and you enter a variable name by itself, Python prints its value.

Once we get to script programming, you'll find variables to be essential. But we can use variables now to build complex expressions.

For example, consider the problem of using the quadratic formula, which is as follows:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Taking a square root is the same operation as raising it to the power of one half (0.5). We therefore have all the tools we need to use this formula.

As you may recall from high school, the quadratic formula solves an equation of the following form:

$0 = ax^2 + bx + c$

One quadratic I've always been fascinated with is the one that determines the golden ratio, in which A/B equals (A + B)/A. One of the properties is that the square of this number is 1 more than the number itself.

$$x^2 = x + 1$$

This gives us a quadratic equation like this:

$$0 = x^2 - x - 1$$

This gives us values for `a`, `b`, and `c`, which we can enter into Python.

```
>>>a = 1
>>>b = -1
>>>c = -1
```

Now let's apply the quadratic formula. First, let's get the determinant, which is the portion of the formula under the square root sign. I'll abbreviate this value as `determ` to make for less typing.

```
>>>determ = (b * b - 4 * a * c) ** 0.5
```

Again, assignment creates a variable, in this case, `determ`. As always, we can get the value of this variable by entering it alone on a line.

```
>>>determ
2.23606797749979
```

Looking back at the full quadratic formula, it's not too hard to plug this into the rest of the formula to get the final answer or, rather, one of them.

```
>>>x = (-b + determ) / (2 * a)
```

This statement creates `x` as a variable by assigning a value to it, and now we can get its value.

```
>>>x
1.618033988749895
```

This result is indeed the golden ratio, to a high degree of precision!

EXERCISES

Exercise 2.3.1. If you look closely at the quadratic formula and the steps we took to get a value, you should see that there is another value possible for `x`. Using a statement similar to the one we just entered, get this second value for `x`. (Hint: if you turn a few pages back, you'll see that the formula uses a plus-or-minus sign, indicating that there are two different solutions.)

Exercise 2.3.2. What is the problem, if any, with the following series of Python statements?

```
he_loves = 10
she_loves = -10
Love = 2
they_love = he_loves * she_loves + love
```

Exercise 2.3.3. Which of the following are valid names for variables?

```
amount
amount55
_amount
1x
y1
2y
n2
```

Interlude

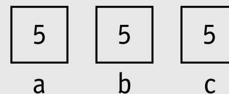
What Do Python Assignments Really Do?

For C, C++, and BASIC, the way I usually define a *variable* is as a named location that stores a value. In other words, a variable is like a little box that has a name on it, into which you can put any value you want as long as it is in the right format.

Such "little boxes," it should be noted, have a series of attributes in C++. A variable, or "box," is declared to only be able to hold certain kinds of data. If I try to put any other data in there, the result is an error.

But Python does things differently. This difference may seem trivial right now. But later in the book, it will matter greatly.

Consider how things are done in other languages. Again, most variables can be viewed as little boxes that contain values for as long as you want them to:



Python instead treats every variable as a *reference*. A reference, in turn, has some similarities to C pointers or to Windows handles. The key point is that when multiple variables are references (that is, refer to) the same value, they do not store the value separately. Let's say several variables are assigned the value 5.

▼ continued on next page